VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Implementace transformací modelů postavených na Petriho sítích v nástroji Kaira

Implementation of Transformations of Petri Nets Based Models in a tool Kaira

2013

Lukáš Tomaszek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Lukáš Tomaszek**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Implementace transformací modelů postavených na Petriho sítích v nástroji Kaira
Implementation of Transformations of Petri Nets Based Models in a tool Kaira

Zásady pro vypracování:

Na katedře je vyvíjen nástroj Kaira. Tento nástroj je určen pro modelování paralelních – distribuovaných aplikací pomocí barevných Petriho sítí. Z těchto modelů pak Kaira umožňuje generovat samostatné aplikace v jazyce C++, které využívají vláken a MPI. Hlavním cílem bakalářské práce bude rozšířit tento nástroj o transformace. Základní myšlenkou je umožnit ve fázi modelování použit konstrukce na vyšší úrovni abstrakce (jako moduly, popsáno v doporučené literatuře), které pak jsou automaticky transformovány na základní konstrukce nástroje Kaira. Cíle bakalářské práce lze shrnout v těchto bodech.
1. Seznamte se s nástrojem Kaira a aktuálním stavem jeho vývoje.
2. Rozšiřte nástroj Kaira o podporu transformací. Při implementaci tohoto rozšíření spolupracujte s Martinem Kozubkem.
3. Navrhněte vhodné transformace a tyto transformace realizujte.
4. Na praktických příkladech demonstrujte funkčnost řešení.

Seznam doporučené odborné literatury:

Stanislav Böhm and Marek Běhálek. 2012. Usage of petri nets for high performance computing. In Proceedings of the 1st ACM SIGPLAN workshop on Functional high-performance computing (FHPC '12). ACM, New York, NY, USA, 37-48. DOI=10.1145/2364474.2364481 http://doi.acm.org/10.1145/2364474.2364481

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Marek Běhálek, Ph.D.**

Datum zadání: 16.11.2012

Datum odevzdání: 07.05.2013

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA * Fakulta elektrotechniky a informatiky * OSTRAVA *

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě dne 19.7.2013 ..... Tomaszek

Abstrakt

Tato bakalářská práce se zabývá rozšířením nástroje Kaira o možnost transformací pomocí modulů. Hlavně pak transformací modulů jako funkce s automatickým ukončením a funkce ukončené uživatelem.

Klíčová slova: funkce, Petriho sítě, Kaira, transformace, moduly

Abstract

This thesis is about extension of the tool Kaira, about posibility to transformed nets using moduls. Especially the transformation modules as a function with automatic end and function end by user.

Keywords: function, Petri nets, Kaira, transformation, moduls

Seznam použitých zkratek a symbolů

MPI	–	Message Passing Interface
GUI	–	Graphic User Interface

Obsah

Seznam obrázků

1 Úvod

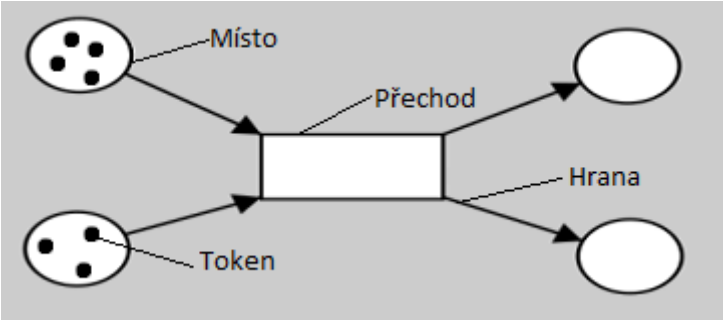
Se zvyšujícím se výkonem výpočetní techniky, budováním superpočítačových center a stále vyšších nárocích na výpočetní výkon, se paralelní aplikace staly nedílnou součástí této doby. Prostředky, na kterých lze spouštět paralelní aplikace, jsou různé. Může se jednat o jeden počítač s více procesory, síť počítačů, či speciální hardware. Nějčastěji dělíme paralelní systémy podle tzv. Flynnovy klasifikace [1].

1.1 Paralelní programování

Chceme-li začít psát paralelní aplikace, musíme vzít v potaz, pro jaký počítač je daná aplikace vyvíjena. Základní otázkou je, jak je organizována paměť vůči výpočetním jednotkám. Rozdělujeme tyto typy: sdílená paměť, distribuovaná paměť a hybridní paměť. Podle typu paměti musíme vhodně zvolit technologii pro přístup k datům a zvolit vhodný synchronizační mechanismus, aby nedocházelo k tzv. deadlockům. Deadlock je programátorská chyba, kdy aplikace nemůže dále pokračovat, protože si výpočetní jednotky blokují potřebná data. Pro výpočty se sdílenou pamětí se využívá semaforů jako synchronizačního mechanismu a jako standard pro programování se využívá OpenMP[2]. U výpočtů s distribuovanou pamětí je pro synchronizaci využíváno zasílání zpráv a pro programování MPI[3]. U hybridních pamětí se využívá kombinace mechanismů a standardů předchozích dvou.

1.2 Petriho sítě

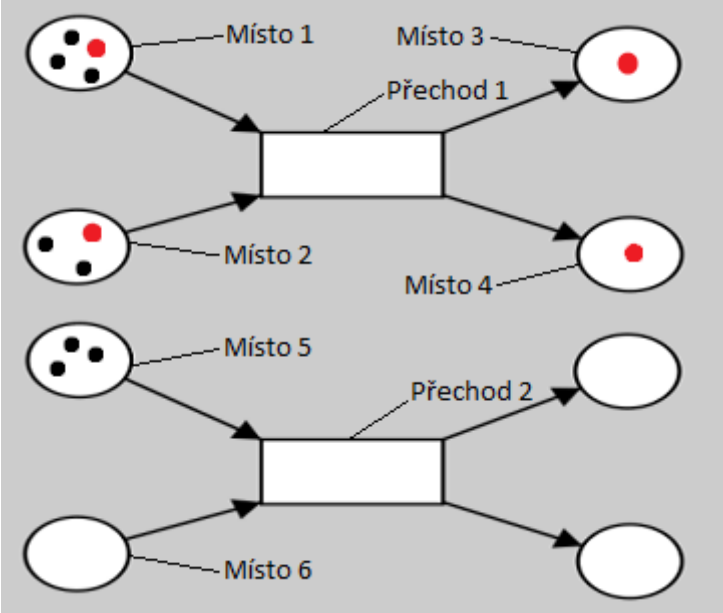
Petriho síť [4], obrázek 1, je orientovaný graf, tvořený místy, přechody a orientovanými hranami. Místa se označují kolečky, přechody obdélníky a hrany šipkami. Hrana může být vedena pouze z místa do přechodu nebo z přechodu do místa. Nikdy nesmí být vedena mezi dvěma místy, či dvěma přechody. V místech jsou umístěny tzv. tokeny.



Obrázek 1: Petriho síť

Přechod se může provést(odpálit) pokud všechna místa, která jsou spojena s daným přechodem hranami (a hrany jsou orientované směrem k přechodu), obsahují minimálně jeden token. Na obrázku 2 jsou umístěny dva přechody. Přechod 1 je proveditelný, v místě 1 i v místě 2 se nachází alespoň jeden token. Při provádění přechodu se odebere

libovolný token z místa 1 a libovolný token z místa 2(vyznačeny červeně) a po provedení umístí token do místa 3 a do místa 4. Oproti přechodu 1 je přechod 2 neproveditelný. Podíváme - li se na obrázek, vidíme, že v místě 6, které je spojeno s přechodem 2, není žádný token.



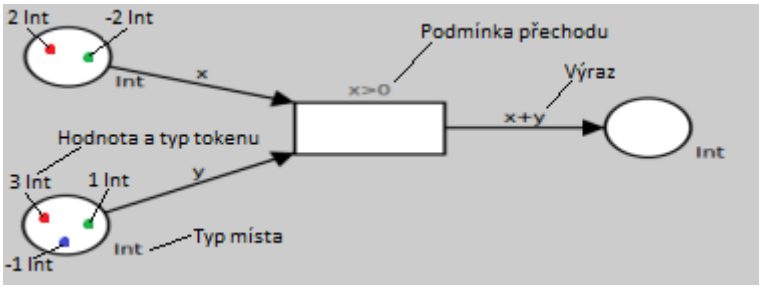
Obrázek 2: Petriho síť - proveditelnost přechodu

Nástroj Kaira využívá barevných Petriho sítí, to je typ Petriho sítí, který je rozšířen o:

- rozdělení tokenů pomocí barev,
- typy,
- podmínky nad přechody,
- hranové výrazy,
- počáteční značení.

U barevných Petriho sítí, obrázek 3, má každý token svou barvu. Při provádění přechodu musí mít všechny vstupující tokeny stejnou barvu. Každý token má dále svůj typ. Typ je přiřazen také místům. V místě se pak nacházejí tokeny, které mají stejný typ, jako toto místo. Dále může být u barevných sítí k přechodu přidána booleovská podmínka. Do této podmínky se před provedením dosadí hodnoty a přechod se provede pouze pokud je podmínka pozitivně vyhodnocena. Podmínka může obsahovat konstanty nebo proměnné. Dalším rozšířením jsou hranové výrazy. Každý výraz udává hodnoty, které vstupují (vystupují) do (z) přechodu. Stejně jako u podmínky může výraz obsahovat proměnné či konstanty. Posledním rozšířením je počáteční značení, které určuje inicializační

hodnoty Petriho sítě. Ke každému místu jsou na začátku přiřazeny tokeny s hodnotami a typy.



Obrázek 3: Barevná Petriho síť

Na obrázku 3 vidíme síť, s jedním přechodem. Modrý token je pouze jeden, v jednom ze vstupních míst, tudíž se přechod pro modrou barvu nemůže provést. Zelené tokeny jsou sice v obou vstupních místech, avšak pro tento případ je x záporné. A podmínka pro splnění přechodu je: x>0. Tudíž i pro zelenou barvu se přechod nemůže provést. Přechod se provede pouze pro červené tokeny. Ve výstupním místě se po provedení přechodu objeví jeden červený token, který byde typu Int a jeho hodnota bude rovna součtu vstupních tokenů. Tedy 5.

2 Kaira

Kaira [5] je nástroj vyvíjený na katedře sloužící k modelování a simulaci paralelních aplikací pomocí barevných Petriho sítí. Z těchto modelů pak Kaira umožňuje generovat samostatné aplikace v jazyce C++. V následující kapitole si popíšeme tento nástroj a části potřebné k této práci. Nástroj se stále vyvíjí a pro novější verze programu se mohou části programu lišit. Tato práce byla vytvářena při verzi Kairy 0.5 a všechny popisované části jsou pro tuto verzi. Pro novější verze doporučuji navštívit stránky Kairy [6].

2.1 Instalace Kairy

Kaira 0.5 podporuje pouze Linuxové distribuce a pro běh programu je zapotřebí těchto knihoven:

- Python 2.6 or 2.7,
- pyGTK,
- pyGTK sourceview2,
- pyparsing,
- scons,
- g++,
- matplotlib.

V distribuci Ubuntu můžeme tyto knihovny nainstalovat v příkazové řádce pomocí příkazu:

- apt-get install python-gtksourceview2 python-pyparsing scons g++ python-matplotlib.

Pro tvorbu samotných zdrojových kódu je dále zapotřebí implementace MPI(například OpenMPI - instalace: apt-get install openmpi-bin openmpi-doc libopenmpi-dev).

Máme-li vše nainstalované můžeme Kairu začít používat pomocí 4 následujících kroků:

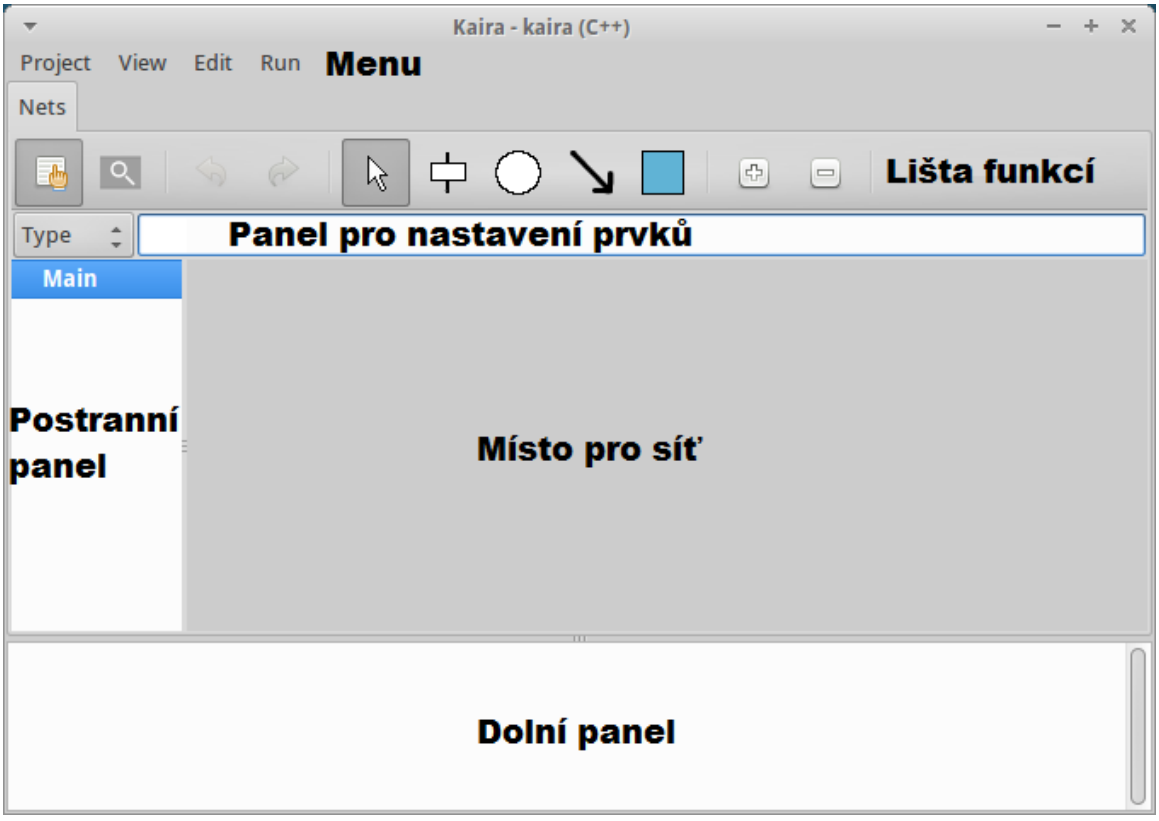
1. stáhnutí instalačního balíčku,
2. rozbalení,
3. sestavení(./build.sh),
4. spuštění(./start.sh).

2.2 Seznámení s Kairou a jejím GUI

GUI Kairy lze rozdělit na několik částí:

- menu,
- lišta funkcí,
- panel pro nastavení prvků,
- postranní panel,
- místo pro síť,
- dolní panel.

Jednotlivé části můžeme vidět na obrázku 4.



Obrázek 4: Kaira

2.2.1 Menu

Hlavní menu nabízí všechny funkce programu. V záložce projekt může vytvářet, otevírat a ukládat nové projekty a otevírat tracelogy. Záložka view nám nabízí nastavení mřížky sítě, pro snadnější pozicování prvků sítě. Další záložka edit slouží k nastavování projektu, hlavičky a testů. Zde bych upozornil na možnost vkládání funkcí(project detail - function), kde můžeme vytvořit svojí vlastní funkci v jazyce C, kterou poté můžeme v síti volat. Poslední záložka run slouží k sestavení kódu a spouštění simulace sítě.

2.2.2 Lišta funkcí

V liště funkcí se nachází 11 tlačítek. První zleva slouží pro zobrazení a skrytí postranního panelu. Druhé pro zobrazení a vypnutí trasování. Následují tlačítka zpět a vpřed. V další části se nachází tlačítka pro vkládání prvků(přechodů, míst, hran a modré oblasti) do sítě. Přechody, místa a hrany byly popsány v úvodu. Jsou to základní prvky Petriho sítí. Modrá oblast v nástroji Kaira slouží pro vykonání stejné části sítě na více procesorech. Poslední dvě tlačítka slouží pro zvětšení a zmenšení sítě.

2.2.3 Panel pro nastavení prvků

Tento panel se skláda z levého výběrového menu a pravé části pro nastavení. U přechodu můžeme nastavit jeho jméno, které slouží pro přehlednost sítě a podmínku přechodu, která stanoví co musí vstupní tokeny splňovat, aby se přechod mohl provést. U místa můžeme nastavit typ tohoto místa a počáteční značení. Počáteční značení se zadává do hranatých závorek a jednotlivé hodnoty se oddělují čárkou. Můžeme zde použít i notaci s dvěma tečkami, např. 0..6. To znamená, že v tomto místě budou hodnoty od 0 po 6. Pro modrou oblast nastavujeme pouze hodnoty procesorů, na kterých se má daná oblast provést a dále zde můžeme tuto oblast pojmenovat. Posledním prvkem je hrana. Zde uvádíme výrazy vstupující(vystupující) z(do) přechodu. Jednotlivé tokeny oddělujeme čárkami. Jsou zde i speciální znaky ~, @ a #. Znak ~vezme všechny tokeny ze vstupujícího místa. V závorce pak můžeme uvést minimální počet těchto tokenů. Znak @ se může používat pouze na hranách, které vycházejí z přechodu a za tímto znakem uvádíme hodnotu procesoru, na který se token umístí. Znak # slouží pro vytváření konstant. # KONSTANTA vytvoříme konstantu, kterou před spuštěním simulace nebo programu musíme nastavit.

2.2.4 Postranní panel

V postranním panelu vidíme všechny sítě a testovací sítě daného projektu. Můžeme zde spravovat všechny sítě a nastavovat zda se mají trasovat či nikoliv.

2.2.5 Místo pro síť

V tomto prostoru můžeme vytvářet samotnou síť. Kliknutím levým tlačítkem myši na jakýkoliv prvek, poté můžeme v liště funkcí nastavit vlastnosti tohoto prvku. Kliknutím

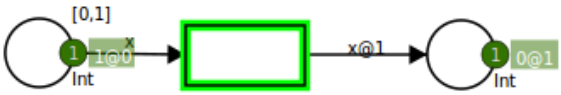
pravým tlačítkem myši se nám poté zobrazí menu. U míst a přechodů máme na výběr, smazání prvku, vypnutí a zapnutí trasování tohoto prvku a možnost vládání kódu v jazyce C do tohoto prvku. U hran pak máme možnost smazání, změny směru hrany, vytvoření obousměrné hrany a přidání bodu. Přidání bodu nám umožní nevytvářet pouze rovné hrany, ale také hrany různých tvarů pro lepší přehlednost sítě.

2.2.6 Dolní panel

Tento panel slouží pro výpisy zpráv a chyb nástroje.

2.3 Simulace v Kaiře

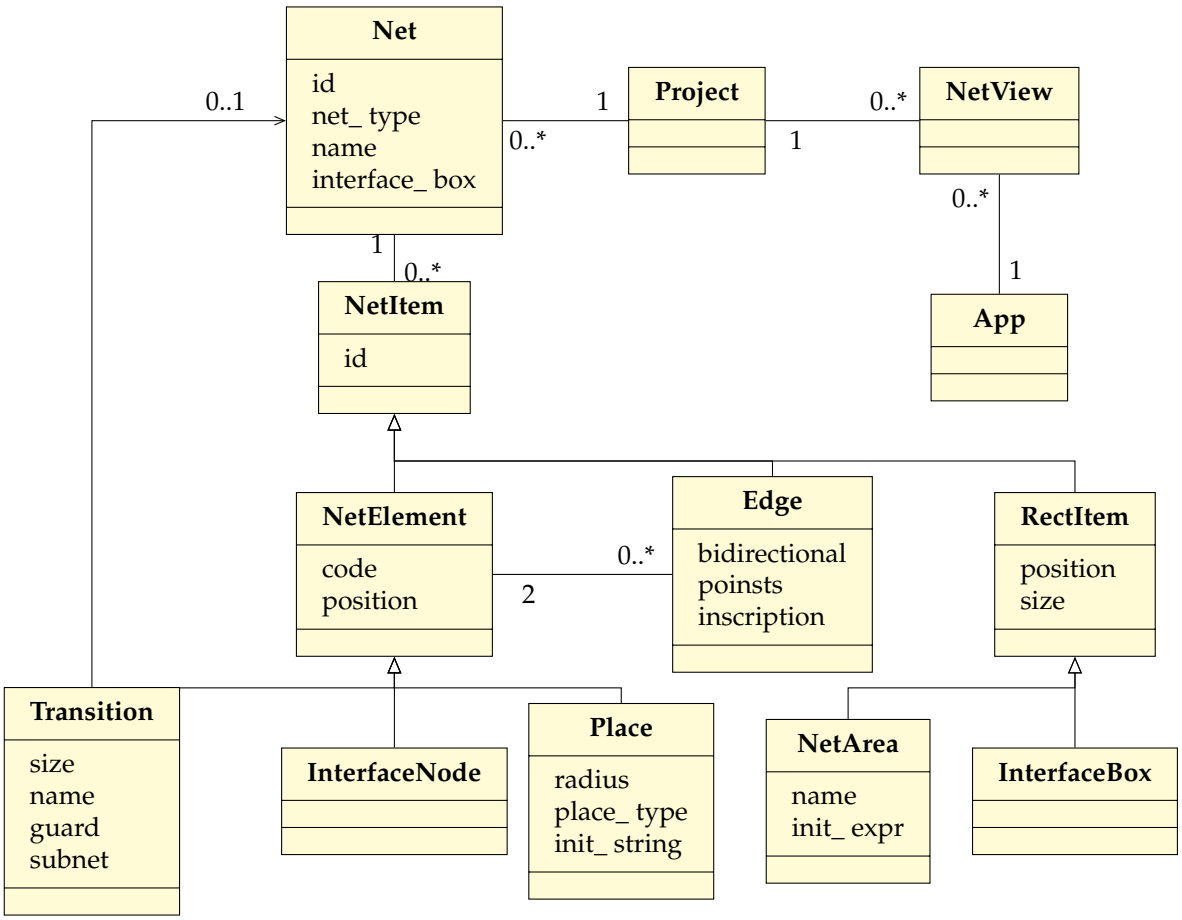
Nyní si ukážeme na velmi jednoduché síti, obr. 5, simulaci. Simulace se nám po zvolení správné nabídky zobrazí v novém okně. Můžeme zde vidět hodnoty tokenů, a na kterých procesorech se zrovna tyto tokeny nachází. Dále pak můžeme odpalovat přechody. Přechody, které se dají odpálit jsou zvýrazněny zelenou barvou.



Obrázek 5: Simulace v Kaiře

2.4 Třídní diagram

Na následujícím obrázku 6 je ukázán třídní diagram s nejdůležitějšími stavy. Je zde zobrazena pouze část, se kterou budeme v této práci pracovat.



Obrázek 6: Třídní diagram

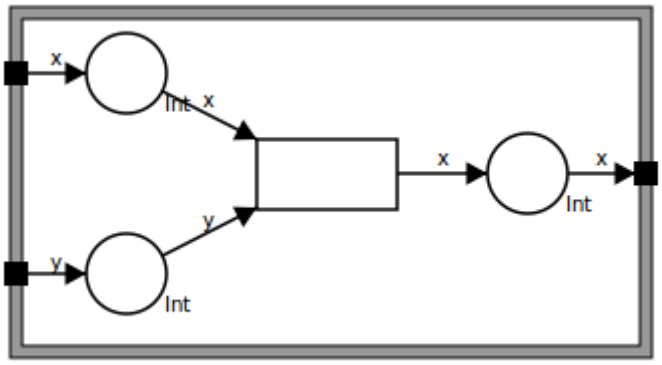
3 Požadavky a vize

Úkolem této práce je rozšířit stávající nástroj Kaira o možnost transformací, navrhnout poté vhodné transformace a ty realizovat.

Transformace nám mají zajistit ulehčení a zjednodušení práce, kdy pomocí vyšších konstrukcí pomocí transformací převedeme síť na nižší konstrukce tohoto nástroje.

Jako možnost ulehčení a zjednodušení práce může být rozdělení Petriho sítí na menší části. V síti do přechodů budeme moci vkládat moduly. Práce se tímto ulehčí. Místo jedné obrovské sítě nám poté bude stačit vytvořit jednu menší a do ní vkládat moduly. Práci tak budeme mít rozloženu na více menších celků, čímž zamezíme vzniku chyb, umožníme znovupoužití často využívaných konstrukcí a celou tvorbu sítě zpřehledníme.

Jak bylo uvedeno v předcházejícím odstavci pro toto rozšíření využijeme modulů, které již byly ve starších verzích implementovány, ale byly z aktuální verze odstraněny. Modul je ohraničená síť, která má vstupy a výstupy. Viz obrázek 7.

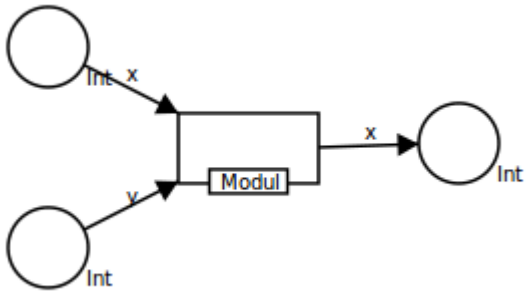


Obrázek 7: Modul

4 Analýza

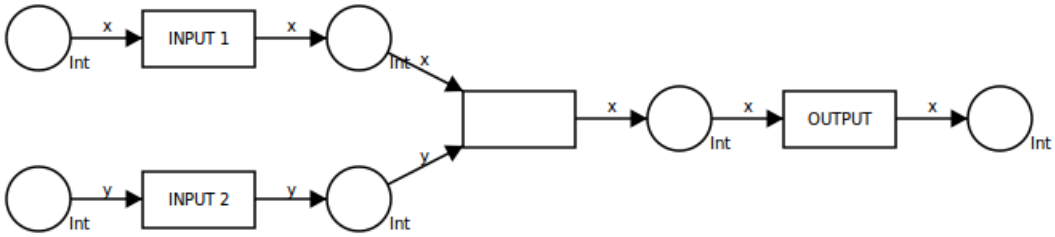
Jako první transformaci, která každého napadne je prosté nakopírování modulu do sítě. Tato transformace bude sloužit pouze pro zjednodušení celé sítě a rozdělení na menší celky. Nebude nijak upravovat novou síť, pouze moduly nakopíruje do původní sítě.

Do sítě, obr. 8, poté vložíme modul, obr. 7. Po transformaci se modul nakopíruje



Obrázek 8: Síť s modulem

do sítě a vznikne tak jedna velká síť, obr. 9, na které budeme moci provádět simulaci a vytvořit zdrojový kód.



Obrázek 9: Transformovaná síť

Podíváme-li se blíže na nově vytvořenou síť, vidíme, že se modul nakopíroval do sítě a přibyly tři nové přechody (INPUT1, INPUT2 a OUTPUT). Tyto přechody slouží jako hranice mezi původní sítí a modulem. Přes tyto přechody do(z) modulu vstupují (vystupují) tokeny. Pro tuto práci budeme předpokládat, že proměnné na hranách do(z) těchto přechodů jsou shodné. Tzn. že uživatel při tvorbě sítě bude muset tohoto docílit(vstupující(vystupující) hrany v modulu a vstupující(vystupující) hrany přechodu s tímto modulem musí mít shodné proměnné).

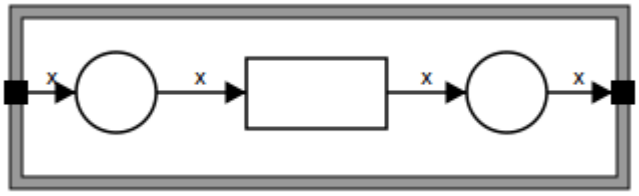
Jak bylo uvedeno tato transformace neprovede žádné změny v původní síti. Tokeny mohou do(z) části, kde je nakopírován modul, vstupovat(vystupovat) kdykoliv. Nyní se podíváme na transformace, kde se modul po transformaci bude chovat jako funkce. Tzn.,

že do této části sítě vstoupí potřebné tokeny, tato část se provede, vystoupí z ní tokeny, a až poté zde budou moci vstoupit další tokeny. Část původního modulu se tedy provede izolovaně od celé sítě a během provádění zde nemohou vstupovat ani vystupovat jiné tokeny a nemohou tak ovlivňovat běh tohoto modulu.

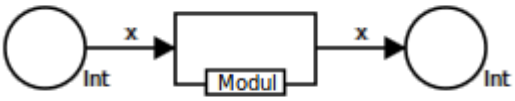
Pokud budeme tedy chtít transformaci, kdy se bude modul používat jako funkce, je třeba vymyslet vhodný mechanismus pro ukončení. Je třeba zjistit, kdy funkce již skončila a dále nepokračuje. V této práci bylo použito dvou mechanismů:

- ukončení uživatelem,
- automatické ukončení.

Nejprve si oba mechanisky ukážeme na práci s 1 procesorem a poté ho rozšíříme na n procesorů. Pro zjednodušení použijeme pouze jednoduchý modul, obr. 10, a vložíme jej do sítě, obr. 11.



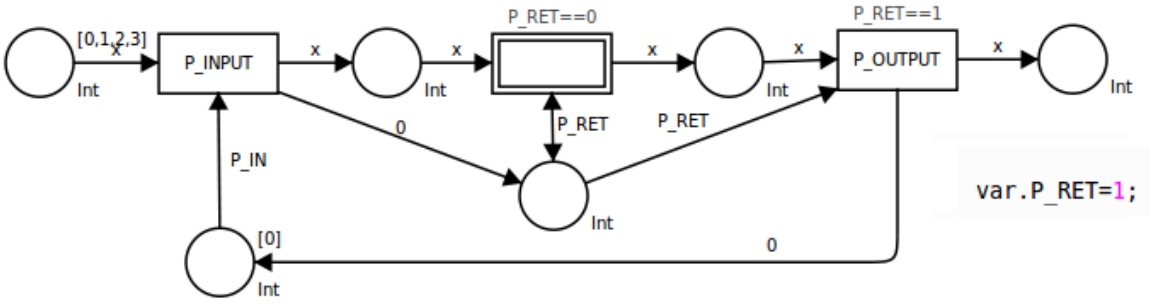
Obrázek 10: Modul



Obrázek 11: Síť s modulem

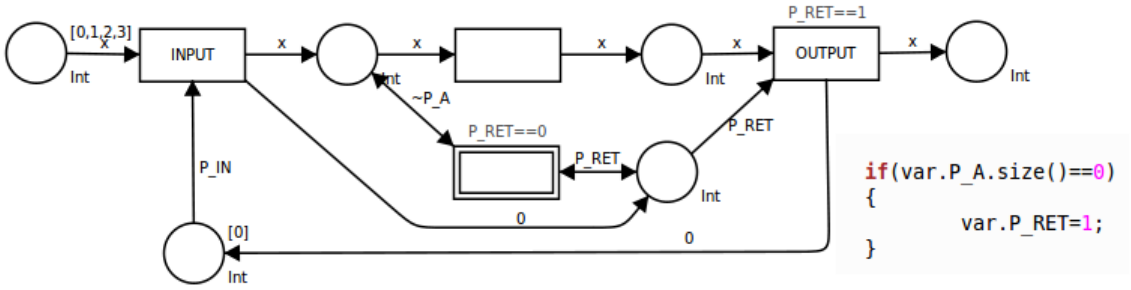
Při ukončení uživatelem budeme předpokládat, že v libovolnám přechodu modulu bude nastavena proměnná P_ RET na hodnotu 1. Výsledná síť, obr 12, poté bude mít navíc dvě místa. První(více vlevo) slouží pro blokování vstupů a druhé uchovává informaci o tom, zda již funkce byla ukončena, či nikoliv. Všechny přechody modulu jsou poté spojeny s druhým místem a provedou se pouze tehdy, má-li proměnná P_ RET hodnotu 0. Tedy pokud funkce ještě nebyla ukončena. Přechod, který nám dává výstupní hodnoty(P_ OUTPUT) má poté podmínku P_ RET == 1. Tady se provede pouze tehdy, je-li funkce již ukončena uživatelem. Dále nám tento přechod dá do prvního místa hodnotu 0, čímž dává najevo, že se funkce může znovu provést.

Pro automatické ukončení nemusí uživatel nastavovat nic. Výsledná síť, obr. 13, kontroluje přechody, zda se mohou provést. Tedy každé místo, které vstupuje do přechodu



Obrázek 12: Transformovaná síť, ukončení pomocí nastavení proměnné P_ RET

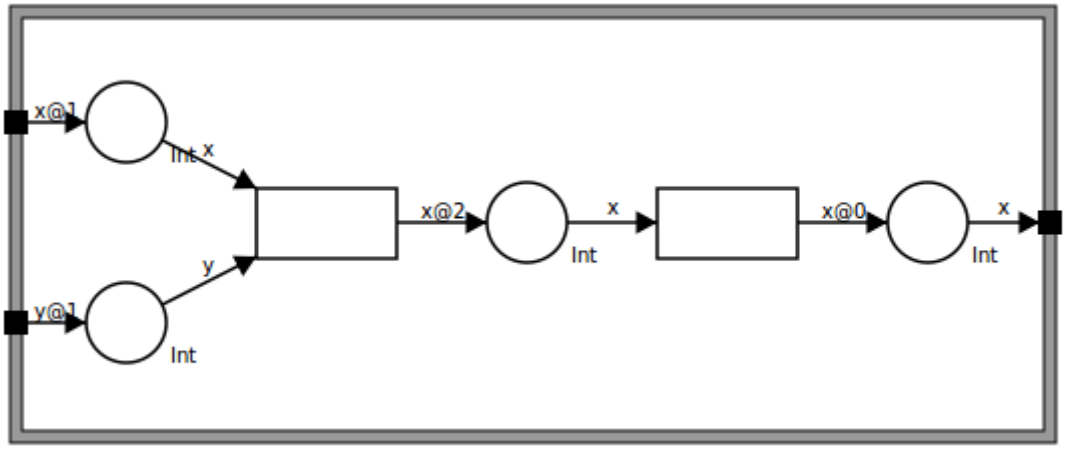
spojíme hranou s novým přechodem, který zkontroluje, zda vstupující místo obsahuje nějaký token. Pokud ano, tento token se vrátí a nic se nezmění. Pokud se však velikost tokenu rovná nule nastaví se hodnota proměnné P_ RET na 1 a stejně jako v minulém příkladě se funkce ukončí.



Obrázek 13: Transformovaná síť, síť se ukončí, pokud nelze provést žádný přechod

Funkce pracující na 1 procesoru se v praxi příliš nepoužijí. Využijeme tedy principu popsaného na jednom procesoru a upravíme pouze mechanismus ukončení. Pro ukázku musíme použít síť, která pracuje na více procesorech, obr. 14. A můžeme ji vložit do sítě v úvodu, obr. 7 na straně 11.

Po transformaci, kdy budeme vyžadovat ukončení uživatelem pomocí proměnné P_ RET dostaneme výslednou síť, která je ukázána na obrázku 15. Princip je stejný jako u transformace s jedním procesorem ukončené uživatelem, která byla popsána výše. Navíc zde přibyl mechanismus, který zajistí, aby byla ukončena práce na všech procesorech. Je zde hlavně využíváno proměnné P_ RET a proměnné P_ COUNT. P_ COUNT udává počet již ukončených procesorů. Na začátku je tato proměnná nastavená na hodnotu 0. Jakmile uživatel na libovolném procesoru nastaví P_ RET na hodnotu 1, tedy že se má funkce ukončit, začne se čekat na dokončení práce ostatních procesorů a postupně se proměnná P_ COUNT inkrementuje. Jakmile je ukončena práce na všech procesorech(P_



Obrázek 14: Modul pracující na více procesorech

COUNT je rovna počtu procesorů pracujících v modulu) funkce se ukončí a vydá výsledek.

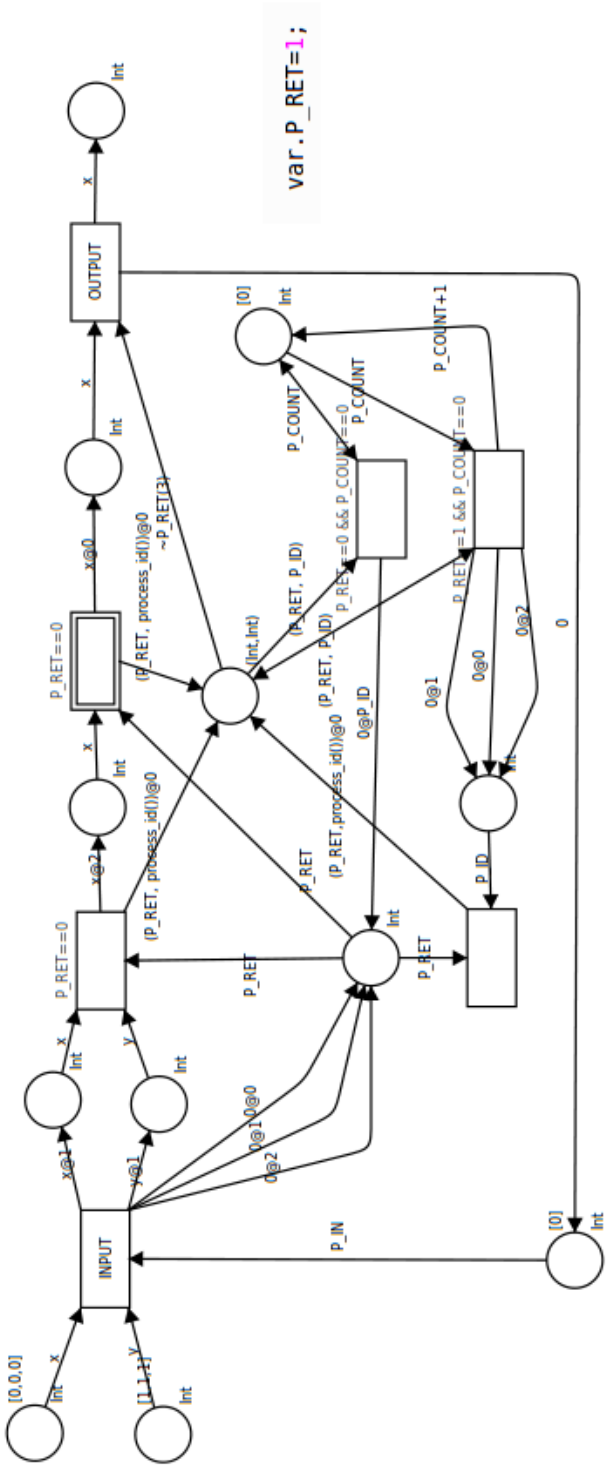
Transformace s automatickým ukončením, když se nemůže provést žádný přechod, na více procesorech, je o něco složitější. Je třeba si zde uvědomit, že pokud se provede libovolný přechod, mohou se tak uvolnit nové tokeny pro jiné procesory. Nestačí tedy počkat až jednotlivé procesory ukončí svou práci, ale při každém provedení přechodu je třeba opětovně kontrolovat všechny procesory, zda nemohou provést některý přechod. Kontrola pak probíhá stejně jako u jednoho procesoru.

Výsledná síť po transformaci na více procesorech a automatickým ukončením pak může vypadat jako je ukázáno na obrázku 16.

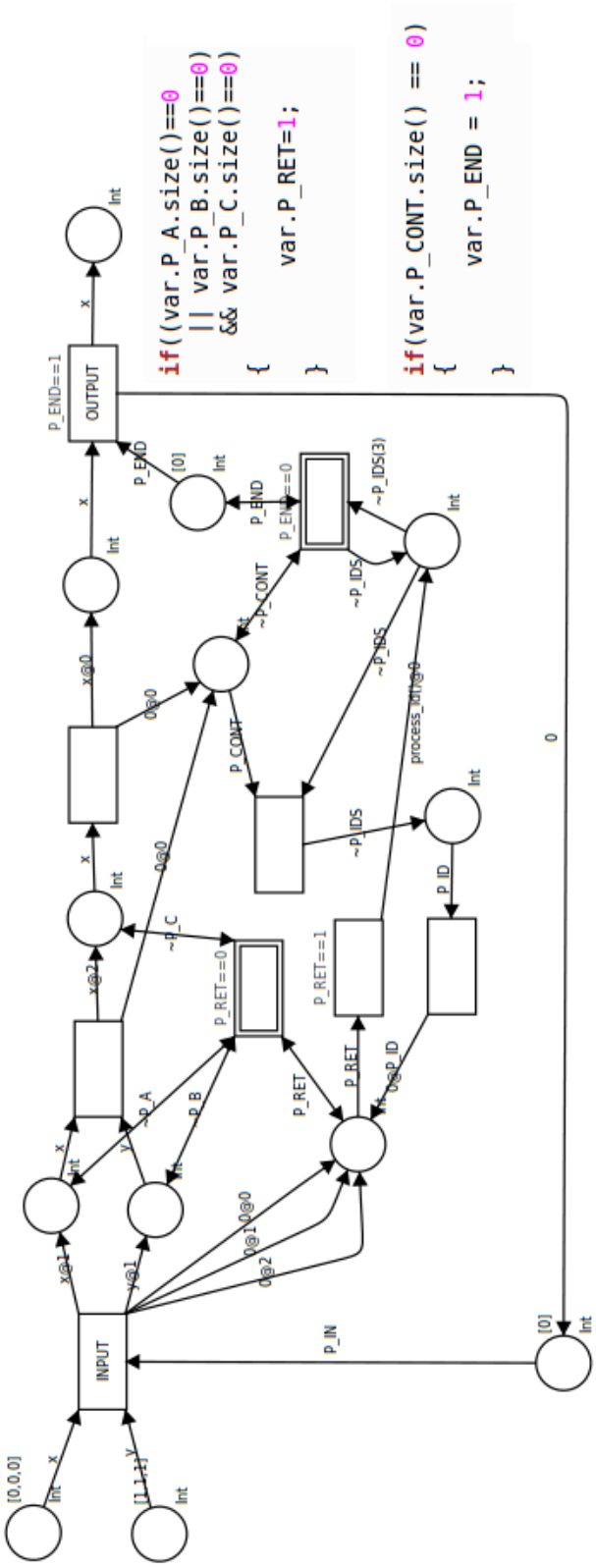
Podíváme-li se na obě transformace na více procesorech, můžeme si všimnout, že zde mohou po předchozím provedení funkce zůstat tokeny, které by poté ovlivňovaly běh funkce, a také se místa ve funkci, které měly nějakou inicializační hodnotu znovu neinicializují. Je tedy třeba ještě po každém provedení funkci opět obnovit. Nejprve odstranit všechny tokeny, které zde zůstaly,obr. 17, a poté inicializovat místa, obr. 18.

Pro vymazání stačí všechna místa v modulu propojit s jedním přechodem, který odebere všechny tokeny.

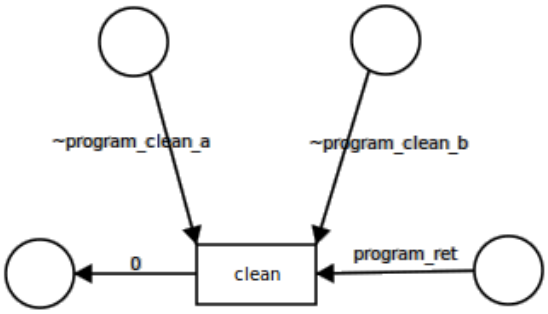
Pro inicializaci jednoho místa je třeba vytvořit nové místo, které bude shodné s původním a poté pomocí přechodu inicializační tokeny nakopírovat.



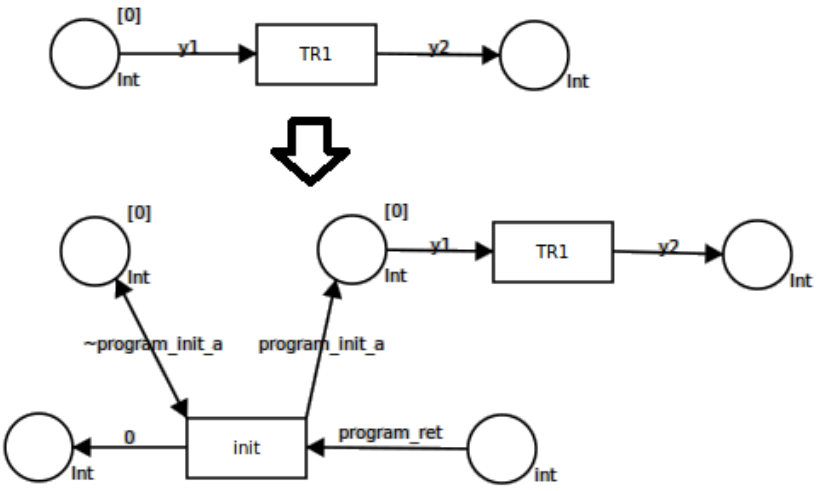
Obrázek 15: Transformovaná síť, pomocí nastavení proměnné P_ RET



Obrázek 16: Transformovaná síť, síť se ukončí, pokud nelze provést žádný přechod



Obrázek 17: Vyčištění míst po provedení funkce



Obrázek 18: Inicializaca míst po provedení funkce

5 Návrh

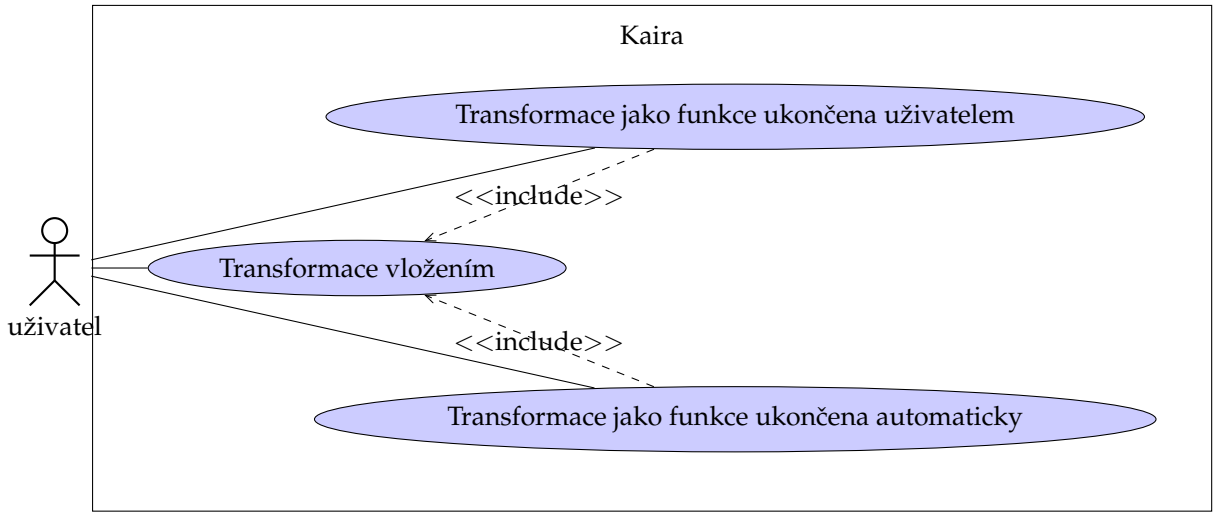
Nejprve si v bodech shrneme, co chceme udělat.

1. Vyhledat ve zdrojových kódech moduly a obnovit jejich funkčnost.
2. Vytvořit možnost transformace prostým kopírováním.
3. Vytvořit možnost transformace modulu jako funkce s ukončením uživatelem.
4. Vytvořit možnost transformace modulu jako funkce s automatickým ukončením.

Moduly již byly naimplementovány v minulých verzích. Je tedy zbytečné je navrhovat. Dále proto bude popis návrhu pouze transformací.

5.1 Diagram případů užití

Na obrázku 19 vidíme diagram případů užití transformací, které budeme implementovat. Tzn. transformaci prostým vložením a transformace jako funkce, které budou využívat prosté vložení pro nakopírování prvků.



Obrázek 19: Diagram případů užití

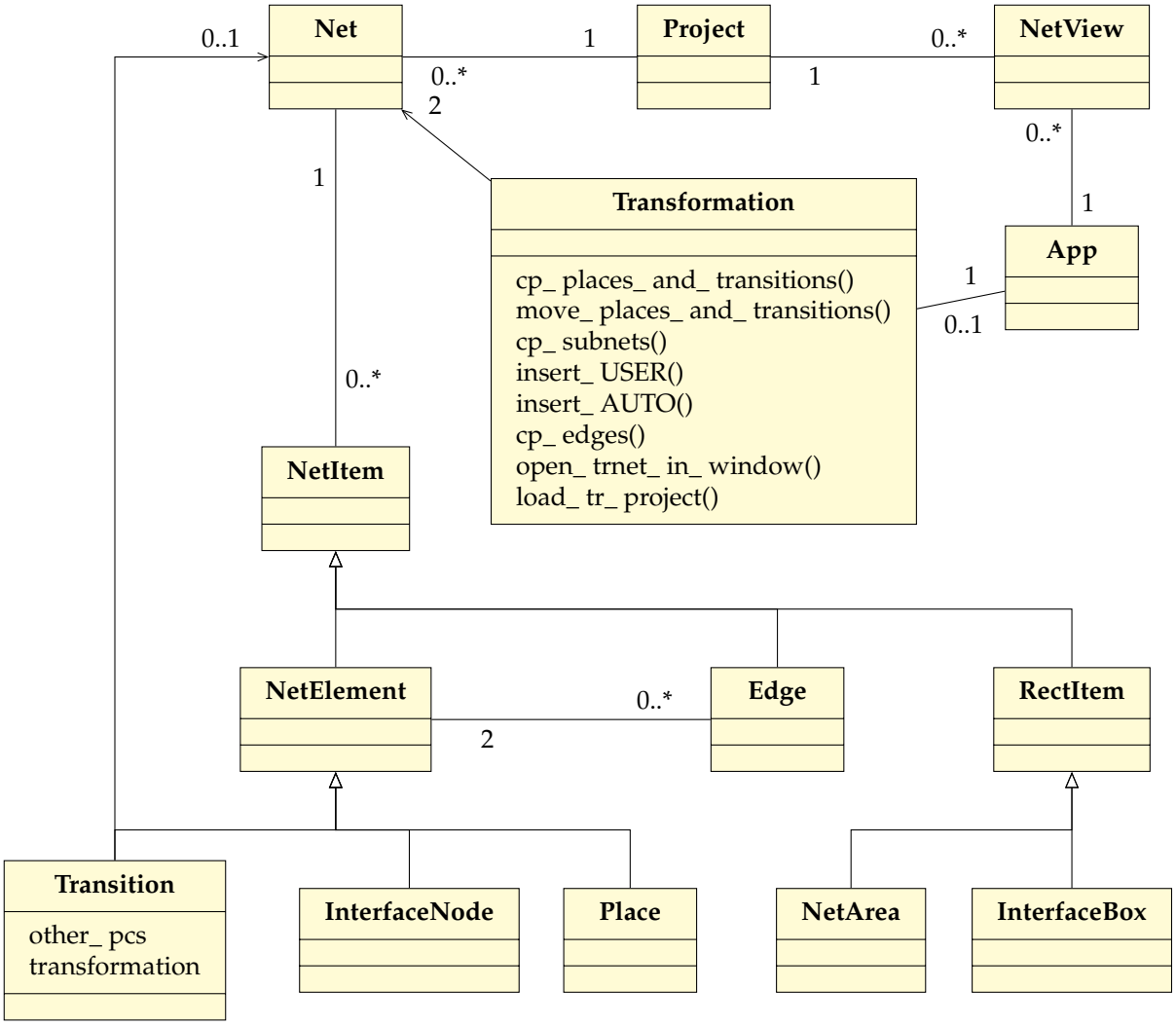
5.2 Třídní diagram

Při programování paralelních aplikací s distribuovanou pamětí se nejčastěji používá jeden procesor pro řízení. V Kaiře se nejčastěji jako tento procesor využívá procesor 0. Podíváme-li se zpět do kapitoly analýzy uvidíme, že pro funkce bylo také využito tohoto procesoru. Budeme proto transformovat sítě tak, aby tento procesor byl řídíci. Dále však potřebujeme znát ostatní procesory, které se v této funkci budou používat. Budeme to

tedy muset v návrhu zohlednit. Dále potřebujeme vědět, jak se má modul transformovat, jestli prostým vložením, či jako funkce.

Pro zobrazení výsledné sítě využijeme dvou možností:

- výslednou síť pouze zobrazíme,
- výslednou síť otevřeme jako nový projekt.



Obrázek 20: Třídní diagram

Do třídy Transition jsme přidali dvě proměnné. První slouží pro uchování hodnot procesorů na kterých se bude modul provádět. Hodnoty se oddělují čárkami a procesor 0 se zde neuvádí. Druhá proměnná slouží pro uchování informace jaká transformace se má provést. Zadáváme zde hodnoty auto nebo user. Auto pro funkci ukončenou automaticky

a user pro funkci ukončenou uživatelem. Pro všechny ostatní hodnoty se bude modul transformovat prostým vložením.

Dále nám přibyla v třídním diagramu třída Transformation, která zodpovídá za transformace. Má odkaz na 2 sítě. Původní síť s moduly a novou síť, kterou vytvoří po transformaci. Nová síť je již pouze se základními konstrukcemi nástroje Kaira.

Ve třídě Transformation je devět základních funkcí. Nyní si popíšeme, co jednotlivé funkce dělají.

cp_ places_ and_ transitions() zkopíruje z původní sítě do nové sítě všechny přechody a místa. Zatím bez modulů a jejich prvků.

move_ places_ and_ transitions() nakopírované prvky přesune tak, aby se vytvořilo místo pro vložení modulů.

cp_ subnets() zkopíruje přechody a místa z modulů.

insert_ USER() vloží k modulům, které se mají transformovat jako funkce s ukončením uživatele potřebné prvky.

insert_ AUTO() vloží k modulům, které se mají transformovat jako funkce s automatickým ukončením potřebné prvky.

cp_ edges() zkopíruje do sítě všechny hrany.

open_ trnet_ in_ window() zobrazí transformovanou síť.

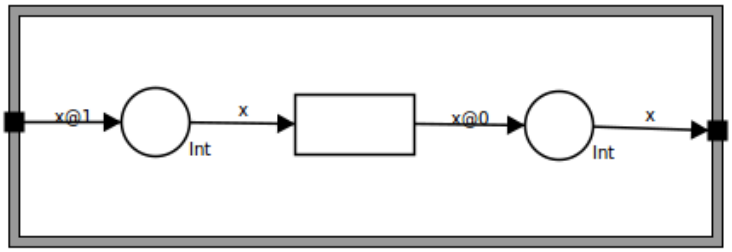
load_ tr_ project() uloží původní síť a otevře transformovanou síť jako nový projekt.

6 Implementace a testování

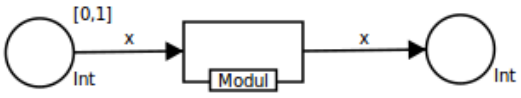
Celá Kaira i s implementovanou částí transformací se nachází na přiloženém CD. Transformování bylo naimplementováno zcela, avšak výsledná síť po transformaci byla velmi nepřehledná(obrázek TransSit.png na přiloženém CD). To bylo způsobeno přechody clear a init. Kdy do přechodu clear musely vést ze všech míst hrany a do přechodu clear musely vést 2 hrany z každého místa, které obsahovalo počáteční značení. Tím se síť zaplnila hranami a v takovémto stavu se dále nedá použít. Transformace jsou tedy naimplementovány bez těchto hran, což nám znemožňuje provést jakoukoliv funkci dvakrát, ale můžeme zde ověřit správnost transformace.

Dalším problémem je, že po vložení více funkcí je nová síť značně rozsáhlá a s každou další transformací se zvětšuje a tím se přehlednost sítě opět zhoršuje.

Nyní tedy vyzkoušíme otestovat, alespoň transformace bez clear a init hran, které by nám výsledek velice znepřehlednily. Mějme tedy modul, obr. 21, a vložme jej do sítě, obr. 22. Výsledek po transformacích pak můžeme vidět na obrázcích 23 a 24.



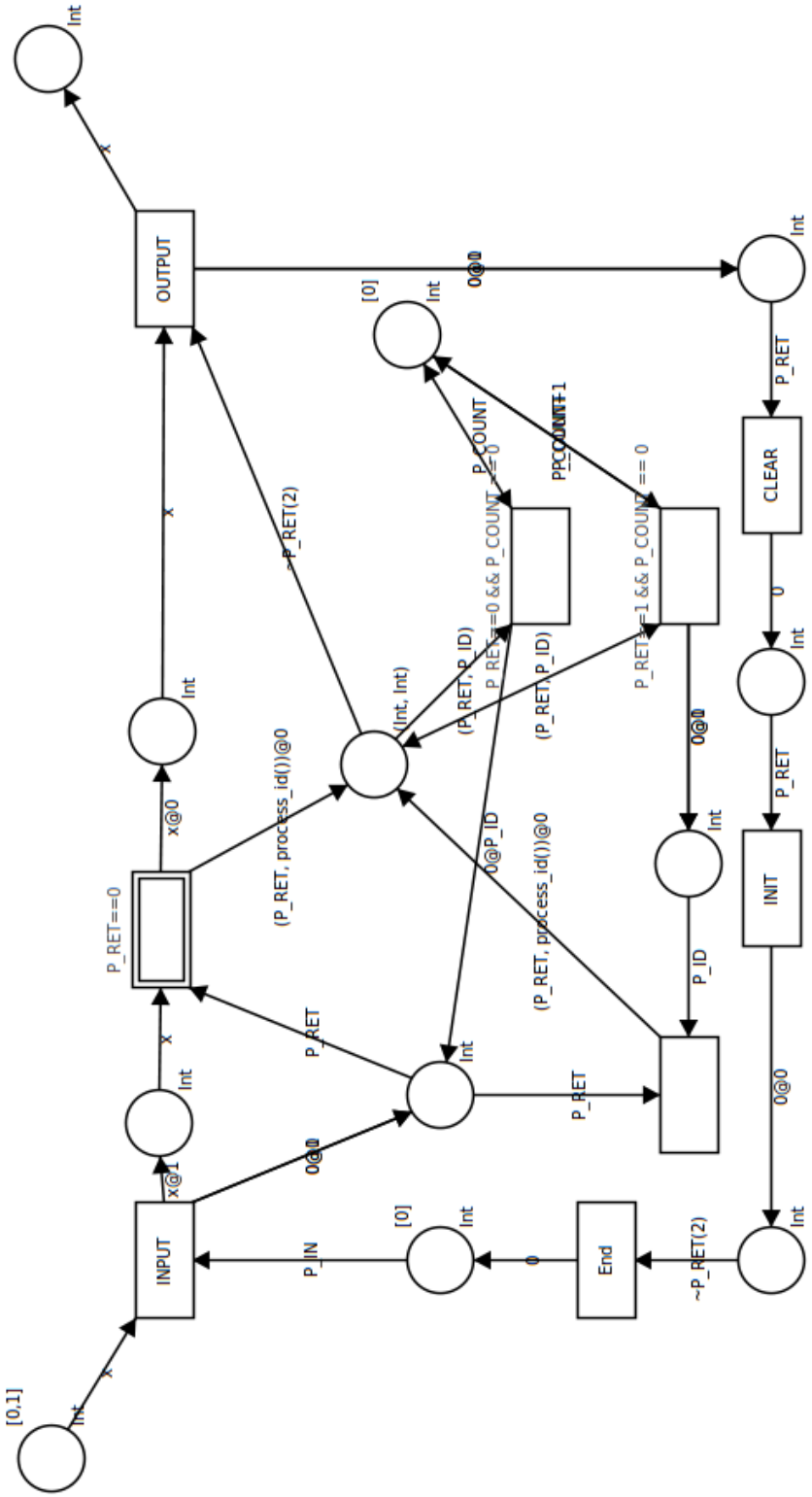
Obrázek 21: Testovací modul



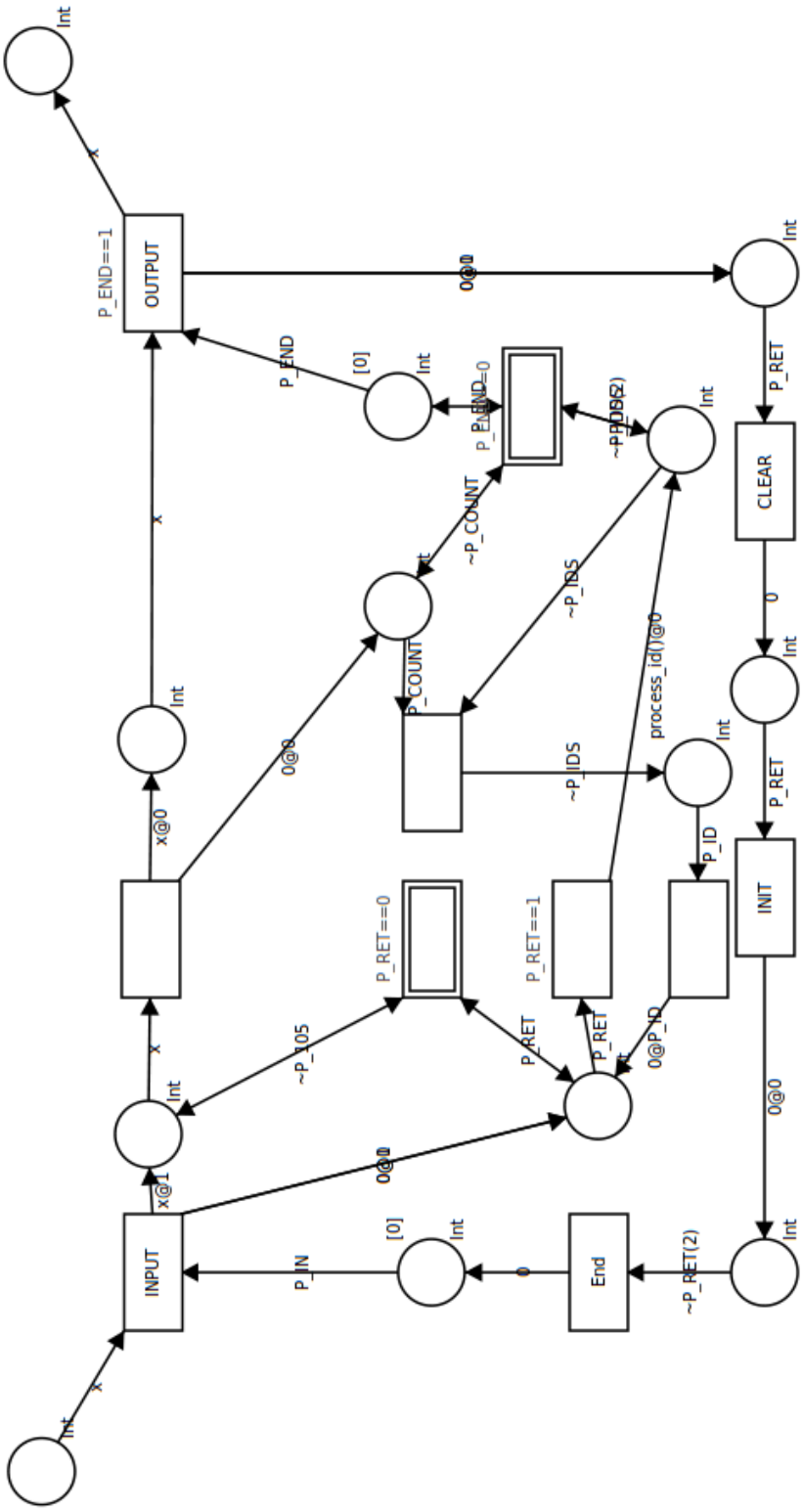
Obrázek 22: Testovací síť

Podíváme-li se na výsledné transformace, jsou shodné s navrhovanými až na pár drobných změn. Přibyly nám zde přechody init a clear, které by měly z funkce odstranit zbylé tokeny a funkci znovu inicializovat. Ovšem z důvodu přehlednosti toto momentálně nedělají. Dále se nám změnil popisek na hranách u funkce s automatickým ukončením. Kdy se využilo pro kontrolu tokenů jiného názvu proměnných. Bylo zde využito id míst a nemuselo se tak programovat s generovanými znaky.

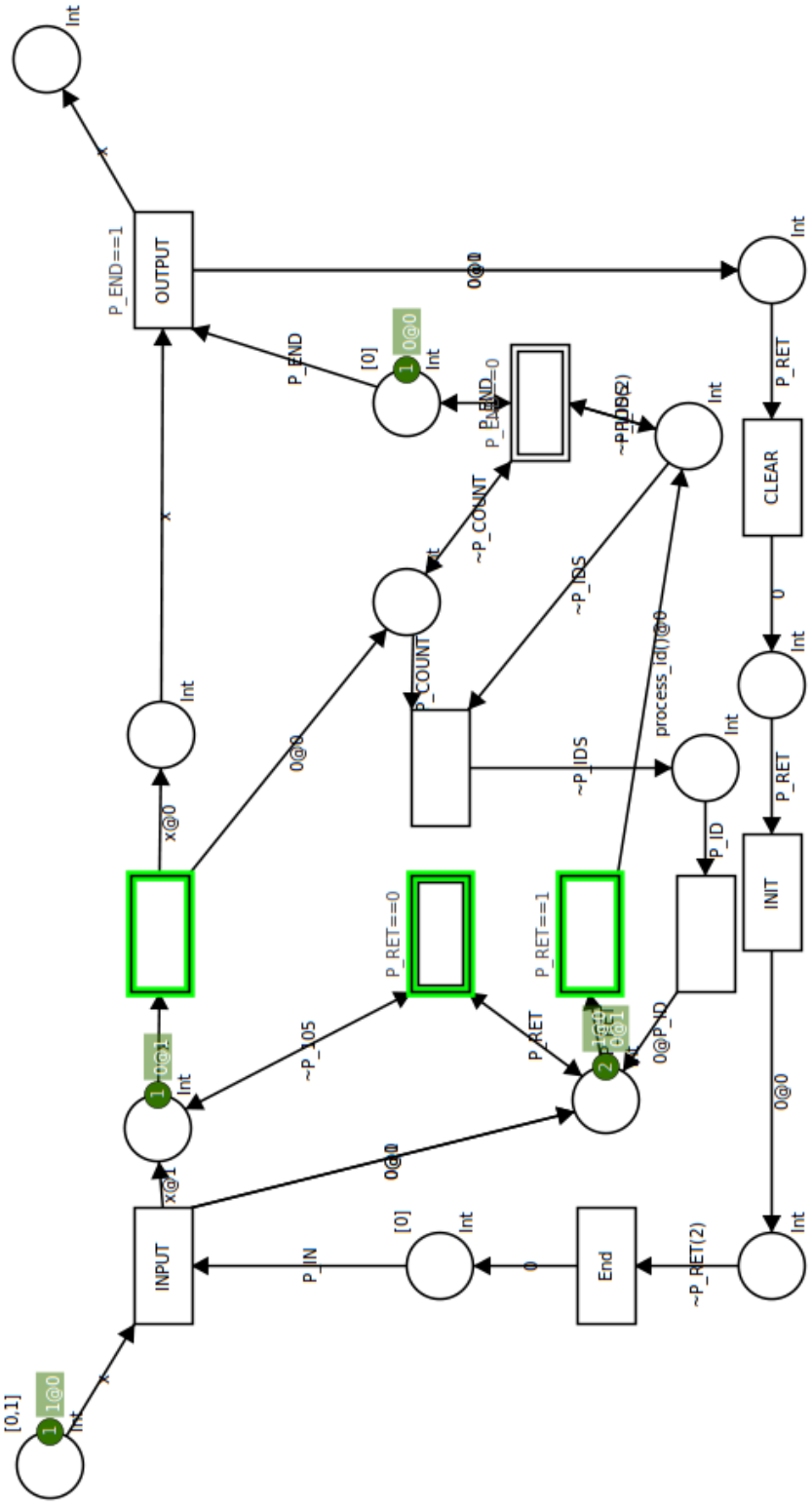
Na obrázku 25 si můžeme poté ověřit, že transformace funguje jako funkce. Zatímco se provádí přechody uvnitř funkce, přechod INPUT se nemůže odpálit a tak zde nepřibydou žádné nové tokeny. Stejně tak bychom si mohli ukázat simulaci na funkci ukončené uživatelem. Oba příklady nalezneme na přiloženém CD ve složce testy.



Obrázek 23: Výsledná transformace jako funkce ukončená uživatelem



Obrázek 24: Výsledná transformace jako funkce ukončená automaticky



Obrázek 25: Ukázka simulace transformace

7 Zhodnocení

Úkolem této práce bylo rozšířit stávající nástroj Kaira o možnosti transformací, kdy se pomocí konstrukcí na vyšší úrovni abstrakce síť pomocí transformací změnila na síť, která využívala pouze základní transformace. K samotným transformacím bylo využito modulů a tato práce se hlavně zaměřovala na vložení modulů do sítě jako funkce.

Transformace, kdy se modul použije jako funkce, se podařilo naimplementovat, avšak výsledná síť nebyla přehledná. Pro možnost testování a ověření správnosti alespoň části transformace byly odebrány hrany spojující přechody init a clear se sítí. Touto úpravou ovšem bylo zamezeno znovupoužití funkce, protože nám zůstávaly volné tokeny a funkce se znovu neinicializovala. Část těchto transformací bez odstraněných hran fungovala podle návrhu.

Úplná funkčnost pomocí základních konstrukcí tohoto jazyka tak, aby nová síť byla přehledná nelze udělat. Jako možné řešení by mohlo být zavedení skrytých hran, které by spojovaly místa s přechody a naopak, avšak by se po transformaci a při simulaci nezobrazovaly. V síti by se poté nemusely zobrazovat hrany vzniklé při transformaci, které spojují místa s init a clear přechody. Transformace jako funkce by poté byly plně funkční. Přestože by se omezily tyto hrany, síť by byla stále velice velká. Nevznikalo by zde ale takové velké křížení hran s prvky a s nově vytvořenou síti by se dalo pracovat. Pokud by ovšem bylo použito více transformací, síť by velmi rychle roztla, a tak by bylo vhodné pro větší počet vkládaných modulů skrývat i prvky, které jsou zde transformací přidány a upravit k tomuto použití simulaci.

Dalším možným řešením by mohlo být simulovat přímo síť s moduly a transformaci použít pouze při vytváření kódu. Transformovat by se mohly pouze moduly, které se vkládají prostým kopírováním a ostatní moduly by zůstaly. Simulace by pak byla upravena tak, aby se chovala jako výsledný kód, který by vznikl po transformacích.

Jako další vylepšení transformací by mohlo být spojení transformace jako funkce s automatickým ukončením s funkcí s ukončením uživatelem. To by zajišťovalo, že pokud by se ve funkci nemohl provést žádný přechod, a uživatel by nenastavil proměnnou P_ RET a použil ukončení uživatelem, funkce by se sama ukončila aby nedošlo k zablokování chodu celé aplikace.

Pokud bychom přestali uvažovat o změně modulu jako funkce, pak by se mohlo využít modulů a transformací k vytvoření knihovny často používaných konstrukcí, či nahradit stávající modrou oblast.

8 Reference

[1] B. Barney, Lawrence Livermore National Laboratory. Introduction to Parallel Computing: https://computing.llnl.gov/tutorials/parallel_comp/. Dostupné 30. 5. 2013.

[2] OpenMP: http://www.openmp.org.

[3] Message Passing Interface Forum: http://www.mpi-forum.org.

[4] J. Markl, Petriho sítě I. Ostrava: VŠB-Technická univerzita Ostrava, 2009. 124 s. (Učební texty).

[5] S. Böhm and M. Běhálek. 2012. Usage of petri nets for high performance computing. In Proceedings of the 1st ACM SIGPLAN workshop on Functional high-performance computing (FHPC '12). ACM, New York, NY, USA, 37-48. DOI=10.1145/2364474.2364481 http://doi.acm.org/10.1145/2364474.2364481

[6] Kaira: http://verif.cs.vsb.cz/kaira/